Mybatis

1、简介

(官方文档:https://mybatis.org/mybatis-3/zh/index.html)

1.1什么是mybatis?

MyBatis 是一款优秀的**持久层框架**

- 它支持自定义 SQL、存储过程以及高级映射
- MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作
- MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，javabean）为数据库中的记录
- MyBatis 本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。2013年11月迁移到Github。

如何获取mybatis?

- maven仓库

```
<!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.6</version>
</dependency>
```

- Github:https://github.com/search?q=mybatis
- 中文文档

1.2什么是持久化

数据持久化

- 持久化就是将程序的数据在持久状态和瞬时状态转化的过程
- 数据库（jdbc） io文件持久化
- 生活:冷藏， 牛奶.

为什么要持久化?

- 有一些对象不能丢掉
- 内存太贵

1.3持久层

Dao层、Service层、Controller层（mvc）

- 完成持久化工作的代码
- 层界限十分明显

1.4为什么使用mybatis?

- 帮助程序员将数据存入数据库中
- 方便
- 更容易上手
- 优点:

```
1.简单易学：本身就很小且简单。没有任何第三方依赖，最简单安装只要两个jar文件+配置几个sql映射文件易于学习，易于使用，通        过文档和源代码，可以比较完全的掌握它的设计思路和实现
2.灵活：mybatis不会对应用程序或者数据库的现有设计强加任何影响。 sql写在xml里，便于统一管理和优化。通过sql语句可以满足操作数据库的所有需求
3.解除sql与程序代码的耦合：通过提供DAO层，将业务逻辑和数据访问逻辑分离，使系统的设计更清晰，更易维护，更易单元测试。sql和代码的分离，提高了可维护性
4.提供映射标签，支持对象与数据库的orm字段关系映射
5.提供对象关系映射标签，支持对象关系组建维护
6.提供xml标签，支持编写动态sql


使用的人多!!!!!!!!!!!!!!
```

配置文件pom.xml-----工具类-----实体类-----mybatis-config.xml-------UserDao--------UserMapper.xml-----UserDaotest

2.第一个Mybatis程序

思路：搭建环境--》导入Mybatis---》编写代码---》测试

2.1、搭建环境

搭建数据库

```sql
drop database if exists mybatis;
create database mybatis;
use mybatis;
create table user(
id int not null primary key,
name varchar(30) default null,
pwd varchar(30) default null
)engine=innodb default charset=utf8;

insert into user values
(1,'张三','123456'),
(2,'李四','123456'),
(3,'王五','123456');

select * from user;
```

新建项目

1.新建maven项目

2.导入依赖

```
<!--mysql驱动 -->
 <dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <version>5.1.47</version>
 </dependency>
 <!--mybatis -->
 <!-- https://mvnrepository.com/artifact/org.mybatis/mybatis -->
 <dependency>
   <groupId>org.mybatis</groupId>
   <artifactId>mybatis</artifactId>
   <version>3.4.6</version>
 </dependency>
 <!--junit -->
 <dependency>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
   <version>4.12</version>
 </dependency>
```

2.2、创建模块

- 编写mybatis的核心配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<!--configuration核心配置文件-->
<configuration>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis?useSSL=false"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="org/mybatis/example/BlogMapper.xml"/>
    </mappers>
</configuration>
```

- 编写mybatis工具类

```
package com.mybatis.utils;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
```

```
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;

//sqlSessionFactory ---> sqlSession
public class MybatisUtils {

    private static SqlSessionFactory sqlSessionFactory;

    static{
        //从 XML 文件中构建 SqlSessionFactory

        //使用mybatis第一步:获取SqlSessionFactory对象
        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream =
Resources.getResourceAsStream(resource);
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //获取sqlSession
    // 既然有了 SqlSessionFactory，顾名思义，我们就可以从中获得 SqlSession 的实
例
    //SqlSession 完全包含了面向数据库执行 SQL 命令所需的所有方法。你可以通过
SqlSession 实例来直接执行已映射的 SQL 语句
    public static SqlSession getSqlSession(){
        return sqlSessionFactory.openSession();
    }

}
```

2.3、编写代码

- 编写实体类

```
package com.mybatis.model;

public class user {
  private int id;
  private String namel;
  private String pwd;


    public user() {
    }

    public user(int id, String namel, String pwd) {
        this.id = id;
        this.namel = namel;
        this.pwd = pwd;
```

```
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNamel() {
        return namel;
    }

    public void setNamel(String namel) {
        this.namel = namel;
    }

    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }

    @Override
    public String toString() {
        return "user{" +
                "id=" + id +
                ", namel='" + namel + '\'' +
                ", pwd='" + pwd + '\'' +
                '}';
    }
}
```

2.4、测试

- Dao接口

```
public interface UserDao {
    List<User> getUserList();
}
```

- 接口实现类由原来的UserDaoImpl转变为一个Mapper配置文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace= 绑定一个对应的Dao/Mapper接口-->
<mapper namespace="com.mybatis.dao.UserDao">

<!--    select 查询语句-->
    <select id="getUserList" resultType="com.mybatis.model.User">
        select * from user;
    </select>
</mapper>
```

- junit测试

```java
    @Test
    public void test(){
        //第一步：获得SqlSession对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        //方式一
        //执行SQL
        UserDao userDao=sqlSession.getMapper(UserDao.class);
        List<User> userList = userDao.getUserList();

        for(User user : userList){
            System.out.println(user);
        }

        //关闭sqlSession
        sqlSession.close();
    }
```

可能会遇到的问题:

1.配置文件没有注册

org.apache.ibatis.binding.BindingException: Type interface com.mybatis.dao.UserDao is not known to the MapperRegistry.

- 注意点:

  org.apache.ibatis.binding.BindingException: Type interface com.mybatis.dao.UserDao is not known to the **MapperRegistry.**

  什么是MapperRegistry?

  核心配置文件(mybatis-config.xml)中注册mappers

```xml
<mappers>
    <mapper resource="com/mybatis/dao/UserMapper.xml"></mapper>
</mappers>
```

2.绑定接口错误

3.方法名不对

4.返回类型不对

```
<mapper namespace="com.mybatis.dao.UserDao">

<!--    select 查询语句-->
<select id="getUserList" resultType="com.mybatis.model.User">
    select * from user;
</select>
</mapper>
```

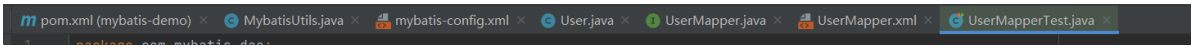
5.Maven导出资源问题

Caused by: java.io.IOException: Could not find resource com/mybatis/dao/UserMapper.xml

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <includes>
      <include>**/*.properties</include>
      <include>**/*.xml</include>
    </includes>
  </resource>
  <resource>
    <directory>src/main/java</directory>
    <includes>
      <include>**/*.properties</include>
      <include>**/*.xml</include>
    </includes>
    <filtering>true</filtering>
  </resource>
</resources>
```

2.5总结

创建过程:



导入依赖---》MybatisUtils工具类---》mybatis-config.xml配置文件---》实体类---》Dao类---》对应的Mapper.xml配置文件---》测试类

3.CRUD

namespace中的包名要和Dao/mapper 接口的包名一致！！！

3.1 select

查询语句：

- **id**:就是对应的namespace中的方法名；
- **resultType**:sql语句执行后的返回值！
- **parameterType**:参数类型

步骤：

1.编写接口

```
    //条件查询
    User getUser(int id);
```

2.编写对应mapper中sql语句

```
<!--    select 查询语句-->
<select id="getUserList" resultType="com.mybatis.model.User">
    select * from user;
</select>

<!--    select条件查询-->
    <select id="getUser" parameterType="int"
resultType="com.mybatis.model.User">
        select * from user where id = #{id}
    </select>
```

3.测试

```
    @Test
    public void select(){
        //获取SqlSession对象
       SqlSession sqlSession= MybatisUtils.getSqlSession();
        //执行SQL
      UserMapper userMapper=sqlSession.getMapper(UserMapper.class);
        System.out.println(userMapper.getUser(1));
    }
```

3.2 insert

```
<!--    insert增加-->
    <insert id="addUser" parameterType="com.mybatis.model.User" >
        insert into user(id,name,pwd) values(#{id},#{name1},#{pwd})
    </insert>
```

3.3 update

```
<!--    update修改-->
    <update id="updateUser" parameterType="com.mybatis.model.User">
        update user set name = #{name} where id = #{id}
    </update>
```

3.4 delete

```
<!--    delete删除-->
    <delete id="deleteUser" parameterType="int">
        delete from user where id = #{id}
    </delete>
</mapper>
```

注意:增删改需要提交事物sqlSession.commit();

3.5 万能map

当我们实体类或者数据库中表的字段、参数过多时，应当考虑使用map!

- UserMapper
- UserMapper.xml
- Test

```
    //增加 map
    int addUser2(Map<String,Object> map);
```

```
<!--    insert增加（map）-->
<!--    对象中的属性，可以直接取出来 传递map的key-->
    <insert id="addUser2" parameterType="map">
        insert into user value (#{id},#{name},#{pwd})
    </insert>
```

```
    @Test
    public void insert2(){
        //获取sqlSession 对象
        SqlSession sqlSession = MybatisUtils.getSqlSession();

        //执行SQL
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        Map<String,Object> map=new HashMap<>();
        map.put("id",4);
        map.put("name","李五");
        map.put("pwd","123456");
        userMapper.addUser2(map);
        sqlSession.commit();//提交事物

        //关闭资源
        sqlSession.close();
    }
```

- map传递参数，直接在sql中取出key即可 【parameterType="map"】
- 对象传递参数，直接在sql中取对象的属性即可 【parameterType="Object"】
- 只有一个基本参数类型的情况下，可以直接在sql中取到

  多个参数用Map,**或者注解!**

3.6 模糊查询

- 在sql拼接中使用通配符%%

```
<!--    模糊查询-->
    <select id="getUserByName" parameterType="String"
resultType="com.mybatis.model.User">
        select * from user where name like "%"#{name}"%"
    </select>
```

```
    //模糊查询
    List<User> getUserByName(String name);
```

4.配置解析

4.1 核心配置文件(mybatis-config.xml)

- mybatis-config.xml
- MyBatis 的配置文件包含了会深深影响 MyBatis 行为的设置和属性信息

- configuration（配置）
  - properties（属性）
  - settings（设置）
  - typeAliases（类型别名）
  - typeHandlers（类型处理器）
  - objectFactory（对象工厂）
  - plugins（插件）
  - environments（环境配置）
    - environment（环境变量）
      - transactionManager（事务管理器）
      - dataSource（数据源）
  - databaseIdProvider（数据库厂商标识）
  - mappers（映射器）

4.2 环境配置(environments)

MyBatis 可以配置成适应多种环境

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。

学会使用多套配置环境

```
<environments default="development">
    <!--第一套环境-->
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <property name="driver" value="com.mysql.jdbc.Driver"/>
            <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useSSL=false"/>
            <property name="username" value="root"/>
            <property name="password" value="root"/>
        </dataSource>
    </environment>
    <!--第二套环境-->
    <environment id="test">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/mybatis?
useSSL=false"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
        </dataSource>
    </environment>
```

Mybatis默认的**事务管理器(transactionManager)**：JDBC　**连接池(dataSource)**：POOLED

4.3 属性（properties)

我们可以通过properties属性来实现引用配置文件

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。

编写db.properties

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mybatis?useSSL=false
username=root
password=root
```

在核心配置文件中引入(注意位置)

```
<!--    引入外部配置文件-->
    <properties resource="db.properties"/>

    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
```

```
                <property name="driver" value="${driver}"/>
                <property name="url" value="${url}"/>
                <property name="username" value="${username}"/>
                <property name="password" value="${password}"/>
            </dataSource>
        </environment>
    </environments>
```

- 可以直接引入外部我呢见
- 可以在其中增加一些属性配置

```
<properties resource="db.properties">
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</properties>
```

- 如果两个文件有同一个字段，优先使用外部配置文件。

4.4 类型别名（typeAliases）

- 类型别名可为 Java 类型设置一个缩写名字
- 它仅用于 XML 配置，意在降低冗余的全限定类名书写

例给User实体类起别名为：user

```
<!--    实体类起别名-->
    <typeAliases>
        <typeAlias type="com.mybatis.model.User" alias="user"></typeAlias>
    </typeAliases>
```

则Usermapper中，resultType="com.mybatis.model.User" 可以换为 resultType="user"

给包起别名：

- 指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean
- 每一个在包中的类，在没有注解的情况下，会使用类的首字母小写的非限定类名来作为它的别名

```
<!--    包起别名-->
    <typeAliases>
        <package name="com.mybatis.model"/>
    </typeAliases>
```

实体类较少时，使用第一种。

否则，建议使用第二种。

第一种可以自己起名，第二种则不行（如果必须要改，可以在实体类上增加注解).

```
@Alias("hello")
public class User {}
```

4.5 设置(setting)

MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为

cacheEnabled	全局性地开启或关闭所有映射器配置文件中已配置的任何缓存。	true | false	true
lazyLoadingEnabled	延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置 `fetchType` 属性来覆盖该项的开关状态。	true | false	false
useGeneratedKeys	允许 JDBC 支持自动生成主键，需要数据库驱动支持。如果设置为 true，将强制使用自动生成主键。尽管一些数据库驱动不支持此特性，但仍可正常工作（如 Derby）。	true | false	False
logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置
mapUnderscoreToCamelCase	是否开启驼峰命名自动映射，即从经典数据库列名 A_COLUMN 映射到经典 Java 属性名 aColumn。	true | false	False

4.6映射器(mapper)

MapperRegistry：注册绑定Mapper文件。

方式一：使用相对于类路径的资源引用

```
<mappers>
  <mapper resource="com/mybatis/dao/UserMapper.xml"/>
</mappers>
```

方式二：使用映射器接口实现类的完全限定类名

```
<mappers>
  <mapper class="com.mybatis.dao.UserMapper"/>
</mappers>
```

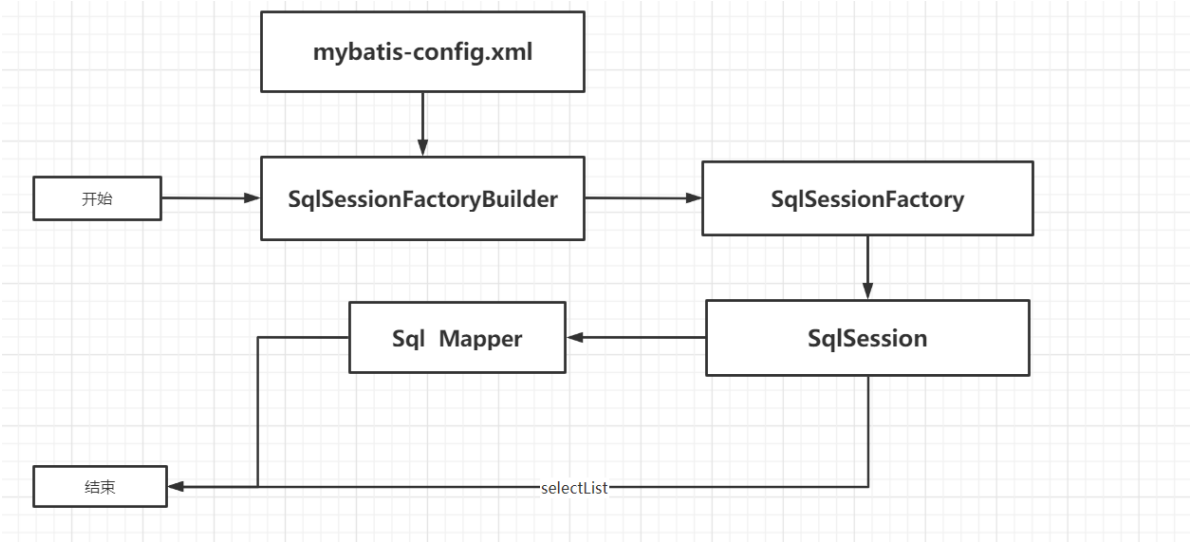
方式三： 将包内的映射器接口实现全部注册为映射器

```
<mappers>
  <package name="com.mybatis.dao"/>
</mappers>
```

注意点(**方式二、三**):

- 接口和他的Mapper配置文件必须同名!
- 接口和他的Mapper配置文件必须在同一个包下!
-

4.7作用域（Scope）和生命周期



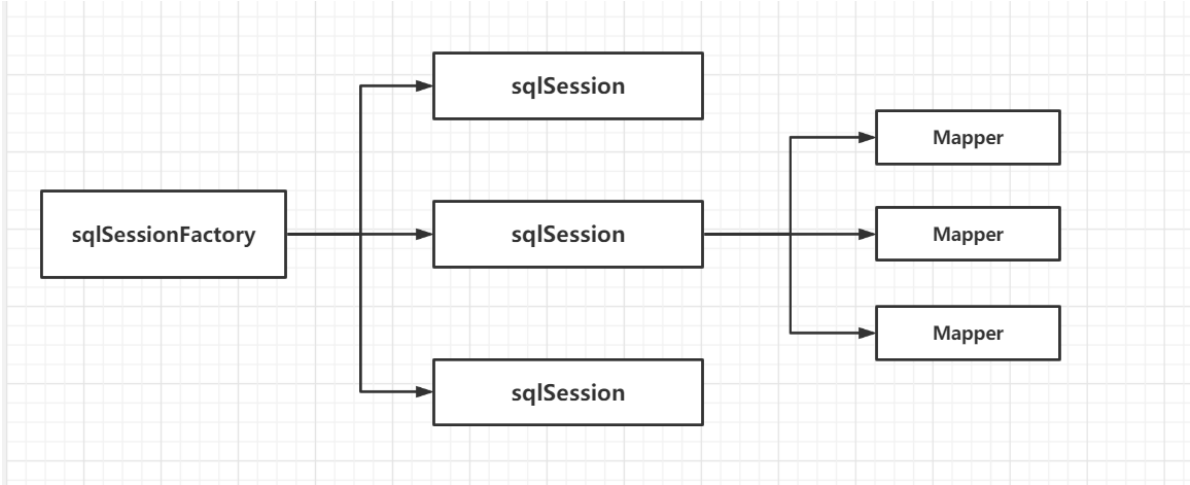
不同作用域和生命周期类别是至关重要的，因为错误的使用会导致非常严重的并发问题

SqlSessionFactoryBuilder:

- 这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它
- 最佳作用域是方法作用域（也就是局部方法变量）。

  SqlSessionFactory(可以想象为数据库连接池):
- SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例
- 最佳作用域是应用作用域, 有很多方法可以做到，最简单的就是使用**单例模式**或者**静态单例模式**

  SqlSession(连接到数据库的一个请求):
- 每个线程都应该有它自己的 SqlSession 实例
- 最佳的作用域是请求或方法作用域
- 用完即关,否则资源被占用!



这里每一个Mapper，都指一个具体的业务!

4.8 其他配置

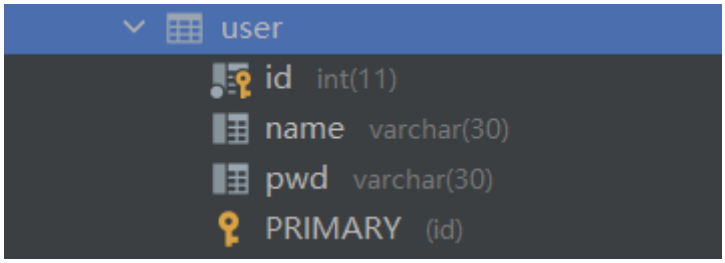
- typeHandlers（类型处理器）
- objectFactory（对象工厂）
- plugins（插件）：

- mybatis-generator-core
- mybatis-plus
- 通用mapper

5.解决属性名和字段名不一致的问题

5.1、问题

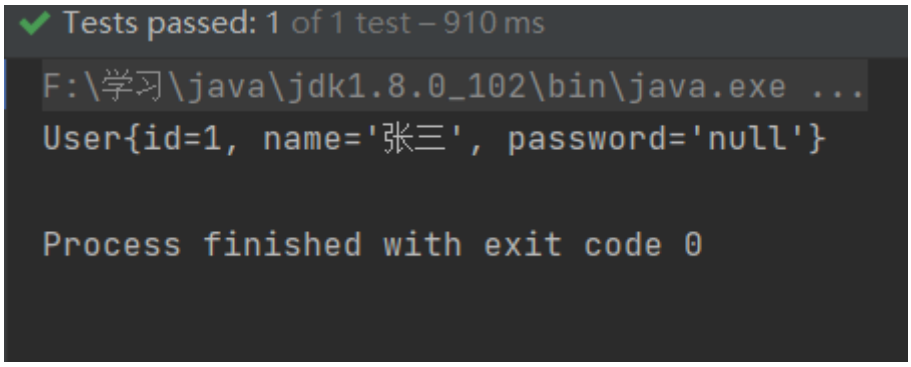
数据库中的字段



实体类中的字段

```
public class User {
  private int id;
  private String name;
  private String password;
```

测试:



解决方法:

- 起别名

```
<select id="getUser" parameterType="int" resultType="user">
    select id,name,pwd as password from user where id = #{id}
</select>
```

5.2、ResultMap 结果集映射

```xml
    <resultMap id="UserMap" type="User">
<!--        column数据库中的字段，property实体类中的属性-->
        <result column="id" property="id"/>
        <result column="name" property="name"/>
        <result column="pwd" property="password"/>
    </resultMap>

    <select id="getUser" parameterType="int" resultMap="UserMap">
        select id,name,pwd as password from user where id = #{id}
    </select>
```

- `resultMap` 元素是 MyBatis 中最重要最强大的元素
- ResultMap 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。
- 可以不用显式地配置它们

6.日志

6.1、日志工厂

当数据库操作出现异常，需要排错时，日志是最好的选择！

原来：sout、debug

现在：日志工厂

logImpl	指定 MyBatis 所用日志的具体实现，未指定时将自动查找。	SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	未设置

- SLF4J
- **LOG4J**
- LOG4J2
- JDK_LOGGING
- COMMONS_LOGGING
- **STDOUT_LOGGING**
- NO_LOGGING

在Mybatis具体用哪一个日志实现，在设置中设定！

STDOUT_LOGGING：标准日志输出

在Mybatis核心配置文件中配置日志。

```xml
<settings>
    <setting name="logImpl" value="STDOUT_LOGGING"/>
</settings>
```

```
Tests passed: 1 of 1 test – 480 ms
F:\学习\java\jdk1.8.0_102\bin\java.exe ...
Logging initialized using 'class org.apache.ibatis.logging.stdout.StdOutImpl' adapter.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
PooledDataSource forcefully closed/removed all connections.
Opening JDBC Connection
Created connection 1613255205.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@60285225]
==>  Preparing: select * from user where id = ?
==> Parameters: 1(Integer)
<==    Columns: id, name, pwd
<==        Row: 1, 张三, 123456
<==      Total: 1
User{id=1, name='张三', password='123456'}
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@60285225]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@60285225]
Returned connection 1613255205 to pool.

Process finished with exit code 0
```

6.2、Log4j

什么是Log4j?

- Log4j是Apache的一个开源项目，通过使用Log4j，我们可以控制日志信息输送的目的地是控制台、文件、GUI组件，甚至是套接口服务器、NT的事件记录器、UNIX Syslog守护进程等
- 可以控制每一条日志的输出格式
- 通过定义每一条日志信息的级别，我们能够更加细致地控制日志的生成过程
- 可以通过一个配置文件来灵活地进行配置，而不需要修改应用的代码。

1.导入Log4j的依赖

```xml
<!--导入log4j依赖-->
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

2.log4j.properties

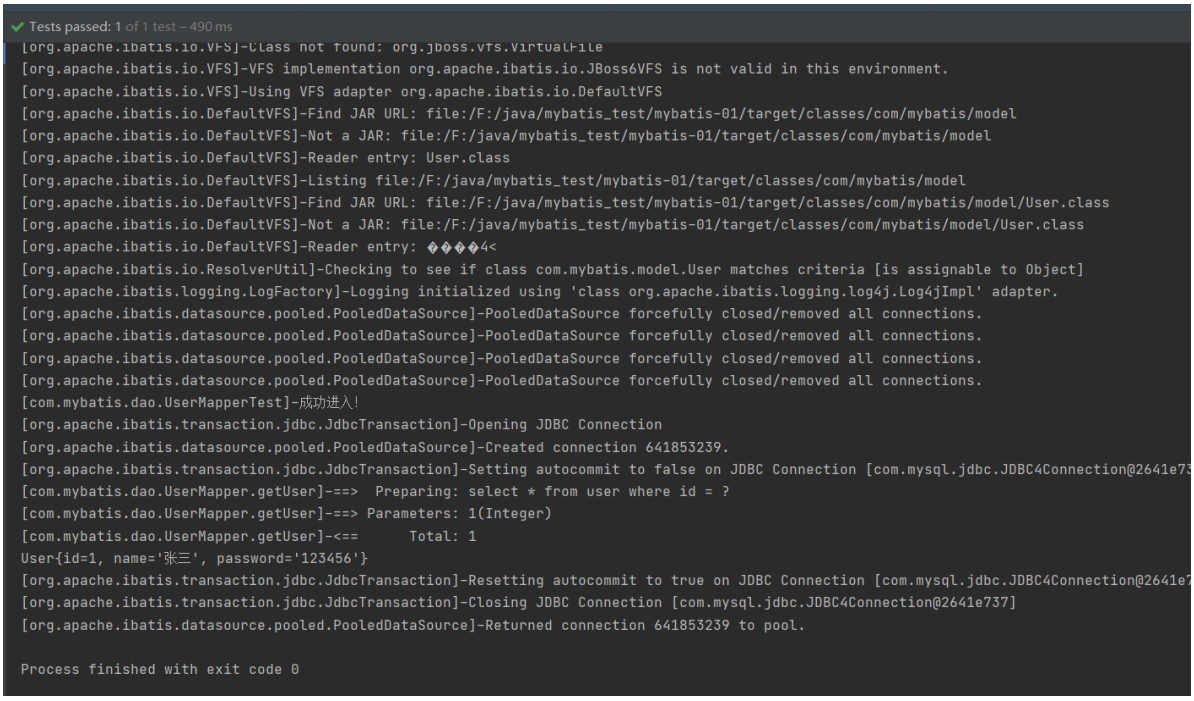
```
#将等级为DE8UG的日去信息输出到console和file这两个目的地。console和file的定义在下面的代码
log4j.rootLogger=DEBUG,console,file

#控制台输出的相关设置
log4j.appender.console = org.apache.log4j.ConsoleAppender
log4j.appender.console.Target=System.out
log4j.appender.console.Threshold=DEBUG
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n

#文存输出的相关设置
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/log4j.log
#log4j.appender.file.MaxFilesize=10mb
log4j.appender.file.Threshold=DEBUG
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[%p][%d{yy-ww-dd}][%c]%m%n
```

```
#日志输出级别
log4j.logger.org.mybatis=DEBUG
log4j.logger.java.sql=DEBUG
log4j.logger.java.sql.statement=DEBUG
log4j.logger.java.sql.ResultSet=DEBUG
log4j.logger.java.sql.PreparedStatement=DEBUG
```

3.配置log4j为日志的实现

```
<settings>
    <setting name="logImpl" value="LOG4J"/>
</settings>
```

4.log4j的使用



简单使用：

1.在要使用log4j的类中，导入包 **import org.apache.log4j.Logger;**

2.日志对象，参数为当前类的class

```
static Logger logger = Logger.getLogger(UserMapperTest.class);
```

3.日志级别

```
logger.info("info--进入log4j");
logger.debug("debug--进入log4j");
logger.error("error--进入log4j");
```

```
F:\学习\java\jdk1.8.0_102\bin\java.exe ...
[com.mybatis.dao.UserMapperTest]-info--进入log4j
[com.mybatis.dao.UserMapperTest]-debug--进入log4j
[com.mybatis.dao.UserMapperTest]-error--进入log4j
```

7.分页

7.1 使用Limit分页

为什么要分页?

- 减少数据的处理量

使用Limit分页

```
语法：select * from user limit startIndex,pageSize
```

使用Mybatis实现分页，核心SQL

1、接口

```
//分页
List<User> getUserLimit(Map<String,Integer> map);
```

2、mapper.xml

```
<!--    分页查询-->
    <select id="getUserLimit" parameterType="map" resultMap="userMap">
        select * from user limit #{startIndex},#{pageSize}
    </select>
```

3.测试

```
@Test
public void userLimit(){
    SqlSession sqlSession = MybatisUtils.getSqlSession();
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    Map<String,Integer> map = new HashMap<>();
    map.put("startIndex",0);
    map.put("pageSize",2);
    List<User> list = userMapper.getUserLimit(map);
    for(User u:list){
        System.out.println(u);
    }
    sqlSession.close();
}
```

7.2 其他分页方式

- RowBounds分页
- 分页插件

了解即可!

8. 使用注解开发

8.1 如何实现

1.注解在接口上实现

```
@Select("select * from user")
List<User> getUsers();
```

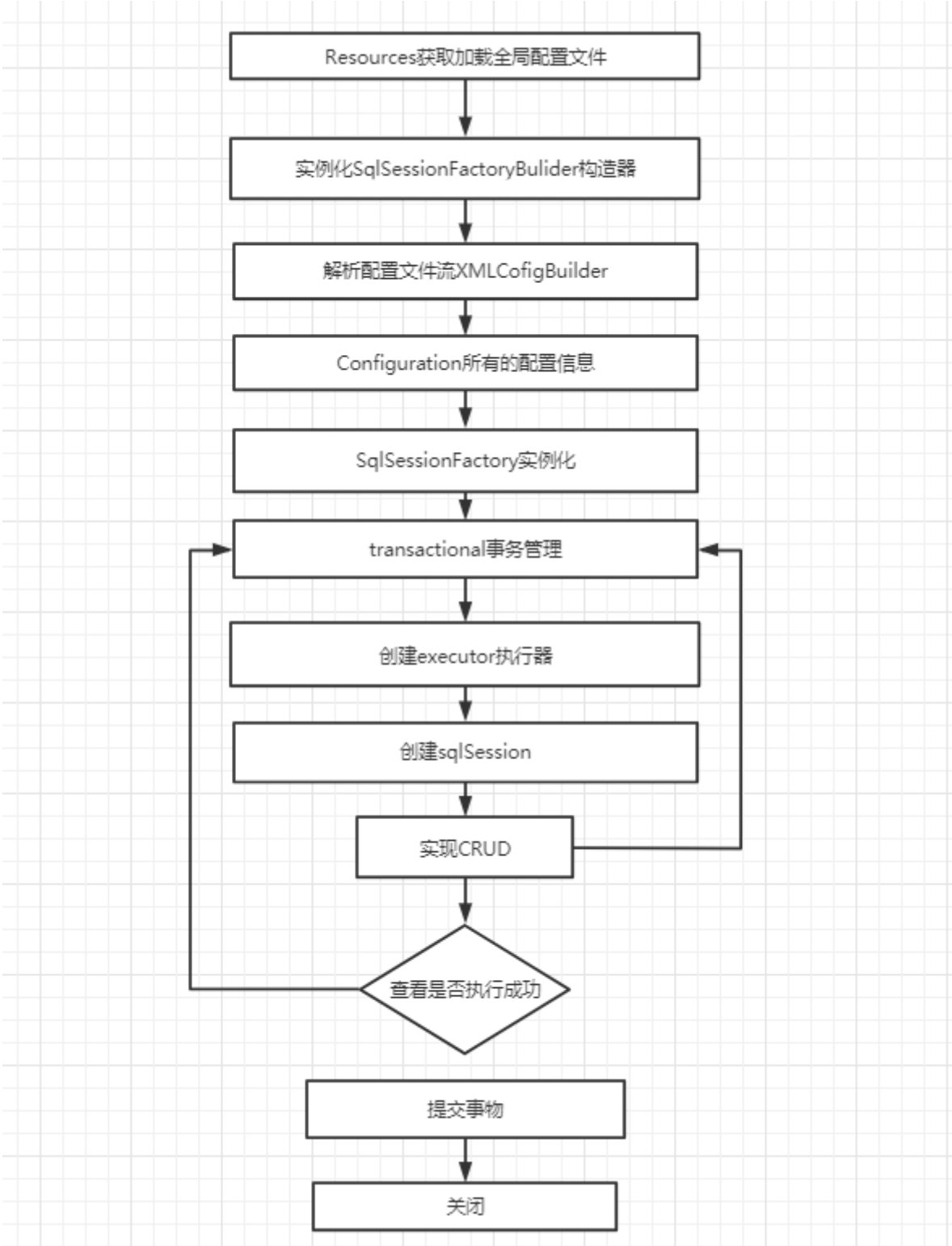
2.需要在核心配置文件中绑定接口

```
<!--绑定接口-->
<mappers>
   <mapper class="com.mybatis.dao.UserMapper"/>
</mappers>
```

本质：反射机制实现

底层：动态代理

Mybatis详细执行流程:



查看：通过**底层代码 Debug**

8.2 CRUD

我们可以在工具类创建的时候自动提交事物！

```
public static SqlSession getSqlSession(){
    return sqlSessionFactory.openSession(true);
}
```

编写接口，增加注解

```
@Insert("insert into user values(#{id},#{name},#{password})")
int addUser(User user);

@Update("update user set name=#{},pwd=#{password} where id=#{id]")
int updateUser(User user);

@Delete("delete from user where id=#[id]")
int deleteUser(@Param("id") int id);

@Select("select * from user where id=#{id}")
List<User> getUsers(@Param("id") int id);
```

测试类

【注意：我们必须将接口注册绑定到我们的核心配置文件中！】

```
<!--绑定接口-->
<mappers>
   <mapper class="com.mybatis.dao.UserMapper"/>
</mappers>
```

关于@Param注解

- 基本类型的参数或者String类型，需要加上
- 引用类型不需要加
- 如果只有一个基本类型的话，可以忽略，但建议加上
- SQL中引用的就是@Param（""）中的值

9.Lombok

简介：

Lombok是一款Java开发播件，使得Java开发者可以通过其定义的一些注解来消除业务工程中元长和繁琐的代码，尤其对于简单的ava模型对象（POJO）。在开发环境中使用Lombok插件后，Java开发人员可以节省出重复构建，诸如hashCodef和equals这样的方法以及各种业务对象模型的accessor和ToString等方法的大量时间，对于这些方法，它能够在编深源代码期间自动帮我们生成这些方法，并没有如反射那样降低程序的性能。

使用步骤：

1.在IDEA中安装Lombok插件

2.在项目中导入lombok的jar包

```
<!--        lombok依赖-->
        <!-- https://mvnrepository.com/artifact/org.projectlombok/lombok -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.20</version>
            <scope>provided</scope>
        </dependency>
```

3.在实体类上加注解即可

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private int id;
    private String name;
    private String password;
}
```

```
@Getter and @Setter
@FieldNameConstants
@ToString
@EqualsAndHashCode
@AllArgsConstructor, @RequiredArgsConstructor and @NoArgsConstructor
@Log, @Log4j, @Log4j2, @Slf4j, @XSlf4j, @CommonsLog, @JBossLog, @Flogger,
@CustomLog
@Data
@Builder
@SuperBuilder
@Singular
@Delegate
@Value
@Accessors
@Wither
@With
@SneakyThrows
@val
@var
experimental @var
@UtilityClass
```

10.多对一处理

10.1 测试环境搭建

1. 导入lombok
2. 新建实体类Teacher, Student
3. 建立Mapper接口
4. 建立Mapper.xml文件
5. 在核心配置文件中绑定我们的Mapper接口或者文件（resource class ）

6. 测试

实体类

```
@Date
public class Student{
    private int id;
    private String name;

    //多个学生可一被同一个老师带
    private Teacher teacher;
}
```

```
@Date
public class Teacher{
    private int id;
    private String name;
    private int sid;
}
```

10.2 按照查询嵌套处理

```
    <resultMap id="UserToTeacher" type="Student">
        <result property="id" column="id"/>
        <result property="name" column="name"/>

<!--        负责的属性，我们需要单独处理 对象：association 集合:collection-->
        <association property="teacher" column="tid" javaType="Teacher"
select="selectTeacherById"/>
    </resultMap>

    <select id="getUsers" resultMap="UserToTeacher">
        select * from student;
    </select>

    <select id="getTeacherById" parameterType="int" resultType="Teacher">
        select * from teacher where id = #{id};
    </select>
```

10.3 按照结果嵌套处理

```
    <resultMap id="UserToTeacher" type="Student">
        <result property="id" column="id"/>
        <result property="name" column="name"/>
        <association property="teacher" javaType="Teacher">
            <result property="name" column="tname"/>
        </association>
    </resultMap>

    <select id="getUsers" resultMap="UserToTeacher">
        select s.id sid,s.name sname,t.name tname
        from student s,teacher t
        where s.tid = t.tid
    </select>
```

11.一对多处理

11.1 测试环境搭建

实体类

```
@Date
public class Student{
    private int id;
    private String name;
    private int tid;
}
```

```
@Date
public class Teacher{
    private int id;
    private String name;

    //一个老师可以教多名学生
    private List<Student> list;
}
```

11.2 按照查询嵌套处理

```
   <select id="getTeacher" resultMap="TeacherStudent">
        select t.id tid,t.name tname,s.id sid,s.name
         from student s,teacher t
         where s.tid = t.id and t.id = #{tid}
    </select>

<!--    集合中的泛型类型，用ofType获取-->
    <resultMap id="TeacherStudent" type="Teacher">
        <result property="tid" column="tid"/>
        <result property="name" column="tname"/>
        <collection property="students" ofType="Student">
```

```
            <result property="id" column="sid"/>
            <result property="name" column="sname"/>
            <result property="tid" column="tid"/>
        </collection>
    </resultMap>
```

11.3按照结果嵌套处理

```
    <select id="getTeacher" resultMap="TeacherStudent">
        select * from teacher where id = #{id}
    </select>

    <resultMap id="TeacherStudent" type="Teacher">
        <result property="id" column="id"/>
        <result property="name" column="name"/>
        <collection property="students" javaType="ArrayList" ofType="Student"
select="getStudentByTeacherId" column="id"/>
    </resultMap>

    <select id="getStudentByTeacherId" resultType="Student">
        select * from student where id = #{id}
    </select>
```

小结

1. 关联 - association 【多对一】
2. 集合 - collection 【一对多】
3. javaType & ofType
    1. javaType 用来指定实体类中属性的类型
    2. ofType 用来指定映射到List或者集合中的实体类型，泛型中的约束条件

面试（关于SQL）：

- Mysql引擎
- InnoDB底层原理
- 索引
- 索引优化

12、动态SQL

什么是动态SQL: 动态SQL就是根据不同的条件生成不同的SQL语句

如果你之前用过 JSTL 或任何基于类 XML 语言的文本处理器，你对动态 SQL 元素可能会感觉似曾相识。在 MyBatis 之前的版本中，需要花时间了解大量的元素。借助功能强大的基于 OGNL 的表达式，MyBatis 3 替换了之前的大部分元素，大大精简了元素种类，现在要学习的元素种类比原来的一半还要少。

```
if
choose (when, otherwise)
trim (where, set)
foreach
```

搭建环境

创建一个基础工程：

1. 导包
2. 编写配置文件
3. 编写实体类
4. 编写实体类对应的Mapper接口 和 Mapper.xml文件

if

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

if语句会在条件（**test**）成立时对SQL语句进行拼接

choose、when、otherwise

有时候，我们不想使用所有的条件，而只是想从多个条件中选择一个使用。针对这种情况，MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG WHERE state = ‘ACTIVE’
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

trim、where、set

```
<select id="findActiveBlogLike" resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

where 元素只会在子元素返回任何内容的情况下才插入 “WHERE” 子句。而且，**若子句的开头为 “AND” 或 “OR”，*where* 元素也会将它们去除**

如果 *where* 元素与你期望的不太一样，你也可以通过自定义 trim 元素来定制 *where* 元素的功能。比如，和 *where* 元素等价的自定义 trim 元素为：

```
<!--prefixOverrides(前缀)      suffixOverrides(后缀)-->
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
<trim prefix="SET" suffixOverrides=",">
  ...
</trim>
```

用于动态更新语句的类似解决方案叫做 *set*。*set* 元素可以用于动态包含需要更新的列，忽略其它不更新的列

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

这个例子中，*set* 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号（这些逗号是在使用条件语句给列赋值时引入的）

foreach

- *oreach* 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量
- 允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符
- 将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象作为集合参数传递给 *foreach*。当使用可迭代对象或者数组时，index 是当前迭代的序号，item 的值是本次迭代获取到的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值

```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT * FROM POST P
  <foreach collection="ids" item="id" open="(" separator="," close=")">
        id = #{id}
  </foreach>
</select>
```

13 缓存

13.1 简介

1. 什么是缓存【Cache】？
    - 存在内存中的临时数据。
    - 将用户经常查询的数据放在缓存（内存）中，用户去查询数据就不用从磁盘上（关系型数据库数据文件）查询，从缓存中查询，从而提高查询效率，解决了高并发系统的性能问题。
2. 为什么使用缓存？
    - 减少和数据库的交互次数，减少系统开销，提高系统效率。
3. 什么样的数据能使用缓存？
    - 经常查询并且不经常改变的数据。

13.2 Mybatis缓存

- MyBatis包含一个非常强大的查询缓存特性，它可以非常方便地定制和配置缓存。缓存可以极大的提升查询效率。
- MyBatis系统中默认定义了两级缓存：**一级缓存**和**二级缓存**
    - 默认情况下，只有一级缓存开启。（SqlSession级别的缓存，也称为本地缓存）
    - 二级缓存需要手动开启和配置，他是基于namespace级别的缓存。
    - 为了提高扩展性，MyBatis定义了缓存接口Cache.我们可以通过实现Cache接口来自定义二级缓存

13.3、 一级缓存

- 一级缓存也叫本地缓存：SqlSession
    - 与数据库同一次会话间查询到的数据会放在本地缓存中
    - 以后如果需要获取相同的数据，直接从缓存中拿，没必要再去查询数据库1

一级缓存失效:

1. 查询不同的东西

2. 增删改操作，可能会改变原来的数据，所以必定会被刷新缓存
3. 查询不同的Mapper.xml
4. 手动清理缓存

```
sqlSession.clearCache();
```

小结：一级缓存是默认开启的，只在一次SqlSession中有效，也就是从 **拿到连接** 到 **关闭** 这个区间!

一级缓存就是一个map。

13.4 、二级缓存

- 级缓存也叫全局缓存，一级缓存作用域太低了，所以诞生了二级缓存
- 基于namespace级别的缓存，一个名称空间，对应一个二级缓存；
- 工作机制
    - 一个会话查询一条数据，这个数据就会被放在当前会话的一级缓存中；
    - 如果当前会话关闭了，这个会话对应的一级缓存就没了；但是我们想要的是，会话关闭了，一级缓存中的数据被保存到二级缓存中；
    - 新的会话查询信息，就可以从二级缓存中获取内容；
    - 不同的mapper查出的数据会放在自己对应的缓存（map）中；

步骤：

1. 开启全局缓存

```
<!--显式的开启全局缓存-->
<setting name="cachEnabled" value="true"/>
```

2. 在要使用缓存的Mapper中开启

```
<!--在当前Mapper.xml中使用二级缓存-->
<cache/>
```

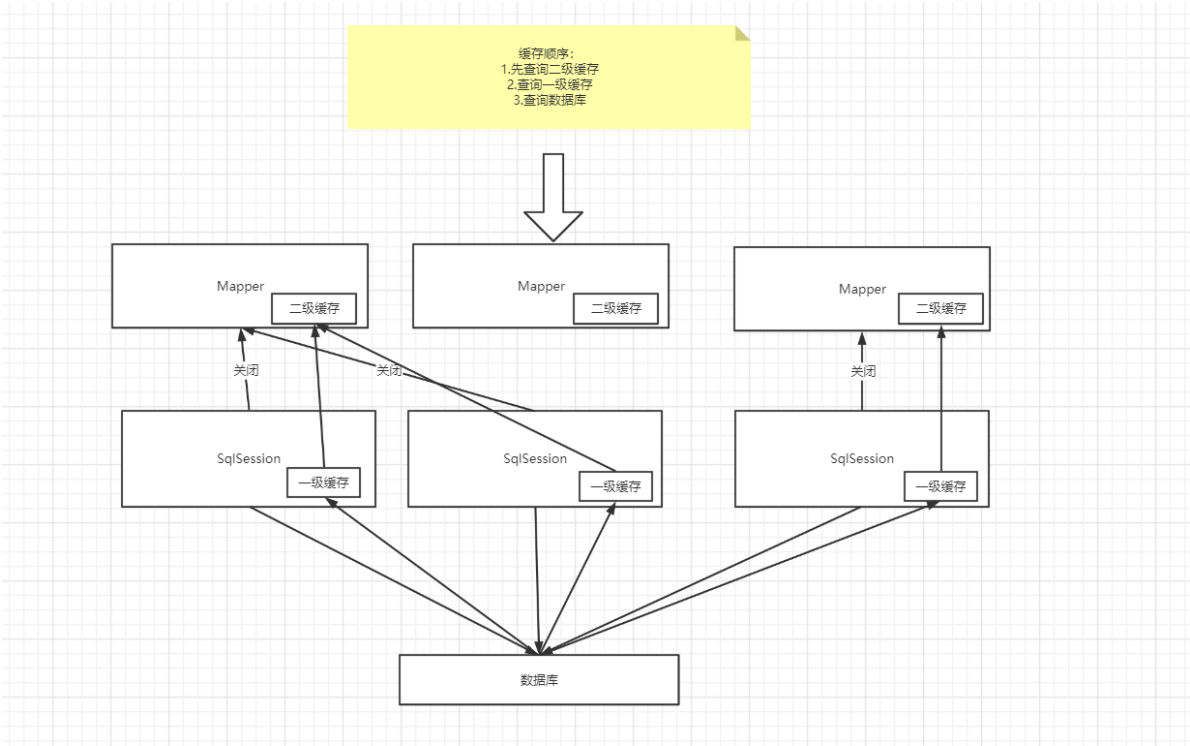
也可以自定义参数

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

注意：

- **当一级缓存关闭后，才会将其本身的缓存丢给二级缓存**
- 需要将实体类序列化

13.5、 缓存原理



13.6、 自定义缓存 - ehcache

要在程序中使用缓存，先导包：

chcache.xml:

```
<?xml version=1.0 encoding=UTF-8?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
         updateCheck="false">

    <diskStore path="./tmpdir/Tmp_EhCache"/>

    <defaultCache
            eternal="false"
            maxElementsInMemory="10000"
            overflowToDisk="false"
            diskPersistent="false"
            timeToIdleSeconds="1800"
            timeToLiveSeconds="259200"
            memoryStoreEvictionPolicy="LRU"/>

    <cache
            name="cloud_user"
            eternal="false"
            maxElementsInMemory="5000"
            overflowToDisk="false"
            diskPersistent="false"
            timeToIdleSeconds="1800"
            timeToLiveSeconds="1800"
```

```
            memoryStoreEvictionPolicy="LRU"/>
/ehcache
```

当前用的比较多的缓存：**Redis**数据库(必须掌握)